Croisière au cœur d'un OS*
Étape "7 et demi" : Gestion de l'espace virtuel utilisateur

Résumé

Après la mise en place d'applications utilisateur basiques et d'appels système à l'article 7, ce neuvième épisode de la série SOS nous amène à traiter de la gestion de l'espace de mémoire virtuelle utilisateur. Au cours de cet article, nous étudierons comment cet espace est géré, quels sont les mécanismes classiques (*demand paging*, *copy-on-write*), et aboutirons à l'implémentation des appels système type Unix habituels `fork()`, `exec()` et `mmap()`.

Introduction

Dans les articles précédents, nous avons choisi de découper chaque espace d'adressage en deux zones : une zone réservée au noyau, et une zone réservée à l'application utilisateur. La première zone est, rappelons-le, identique dans tous les espaces d'adressage, ce qui permet à toutes les applications du système d'avoir la même vue du noyau. Le contenu de cette zone réservée au noyau ainsi que les mécanismes d'allocation permettant sa gestion ont été étudiés à l'article 5 (sous-système *kmem*). Le mécanisme de *synchronisation* de cette zone a quant à lui été étudié dans l'article 7.

En ce qui concerne la zone utilisateur d'un espace d'adressage, nous n'avons pour l'instant prévu aucun mécanisme spécifique. Les applications utilisateur basiques utilisées dans l'article 7 ne peuvent pas allouer dynamiquement de la mémoire, ni utiliser des bibliothèques partagées, ou encore utiliser des appels système tels que `fork()` ou `mmap()`.

La première partie de cet article permettra d'étudier le fonctionnement des mécanismes utilisés classiquement dans les systèmes d'exploitation modernes pour la gestion de la zone utilisateur (partie 1). Nous verrons en quoi ces mécanismes (en particulier le "mapping de fichiers") forment la base pour l'exécution de programmes utilisateur dans les systèmes type Unix. La seconde partie sera consacrée à l'implémentation de ces mécanismes au sein de SOS (partie 2). Les principes exposés en première partie ne seront que partiellement implémentés au cours de cet article. Leur implémentation sera complétée dans les futurs articles de la série.

Les concepts généraux de gestion de mémoire virtuelle utilisés dans SOS sont très proches de ceux utilisés dans KOS ou Linux. L'implémentation proposée est simple, mais aussi moins élégante et moins complexe que celles proposées par exemple dans BSD 4.4, Mach ou Solaris. La gestion de la mémoire virtuelle est un domaine qui a été l'objet de nombreux travaux, publications ou thèses. Le lecteur pourra ainsi consulter la description de la mémoire virtuelle du noyau Linux 2.4 par Mel Gorman [1], la description de la mémoire virtuelle de NetBSD [2] [3], de SunOS [4] [5] ou de Chorus [6].

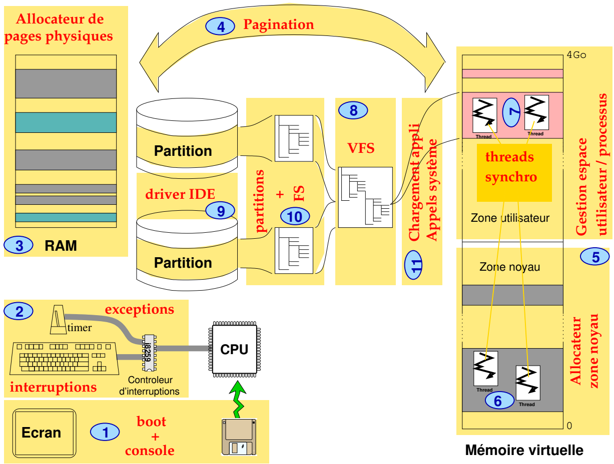

FIG. 1 – Programme des articles

1 Principes

1.1 Découpage de l'espace virtuel

Dans l'article 5, nous avions vu que l'espace de mémoire virtuelle du noyau était découpé en *régions*. Le noyau s'occupait de gérer ces régions pour allouer/libérer de la mémoire virtuelle, et d'associer/de ne pas associer ces régions à des pages en mémoire physique. L'essentiel de la gestion de cet espace consistait à faire un peu d'arithmétique d'intervalles.

Dans SOS, comme dans la plupart des systèmes d'exploitation modernes à mémoire virtuelle, la gestion de la zone utilisateur de chaque espace d'adressage relève des mêmes principes. Cette zone est découpée en différentes *régions virtuelles*. Alors que dans le cas du noyau une région ne pouvait être associée qu'à de la mémoire physique (RAM), dans la partie utilisateur l'approche est plus générale. On peut en effet "mapper" (i.e. associer) en mémoire virtuelle n'importe quelle ressource gérée par le système. Ainsi, quand un programme utilisateur accède aux adresses virtuelles d'une région mappant un fichier par exemple, alors il accédera en fait aux données contenues dans ce fichier. Il y aura bien sûr une étape intermédiaire qui consistera à utiliser la mémoire physique.

Une *région virtuelle* est par conséquent un segment de *pages contiguës* de mémoire virtuelle situé dans la partie utilisateur de l'espace d'adressage. Une *région virtuelle* est caractérisée par :
– son adresse de début et sa taille ;

*La version originale de cet article a été publiée dans GNU Linux Magazine France numéro 72 – Juin 2005 (`http://www.linuxmag-france.org`) et cette version est diffusée avec l'autorisation de l'éditeur.

– ses *droits d'accès* (ou *protection*) : lecture, écriture, exécution ;
– la ressource qu'elle mappe en mémoire virtuelle, ainsi que les indications sur la portion de la ressource qui est mappée.

Il existe deux types de *région virtuelle*, en fonction du type de ressource mappée :
– les régions virtuelles qui sont la *projection en mémoire (ou mapping) d'un fichier*. Accéder à une telle région revient à accéder à une partie d'un fichier stocké sur un système de fichiers (au sens large : stockés sur disque, accessibles par réseau, etc) ;
– les régions virtuelles dites *anonymes*, c'est-à-dire contenant de la mémoire initialement à zéro. Elles ne sont associées à aucun fichier existant physiquement.

Sous Linux, il est possible de lister les régions virtuelles d'un processus donné en lisant le fichier `/proc/<pid>/maps`. Par exemple, pour un processus `bash`, on obtient une liste de régions virtuelles qui ressemble à :

```
08048000-080e6000 r-xp 00000000 03:02 13765      /bin/bash
080e6000-080ec000 rw-p 0009d000 03:02 13765      /bin/bash
080ec000-08147000 rw-p 080ec000 00:00 0
b7da3000-b7dac000 r-xp 00000000 03:02 24311      /lib/tls/libnss_files-2.3.2.so
b7dac000-b7dad000 rw-p 00008000 03:02 24311      /lib/tls/libnss_files-2.3.2.so
...
```

La première région virtuelle est une région représentant une partie du fichier `/bin/bash`, et en lecture/exécution, il s'agit donc vraisemblablement du code de `bash`. La seconde région représente une autre partie du même fichier, à partir de l'offset `0x9d000`, et est en lecture/écriture, il s'agit donc vraisemblablement des données de `bash`. La troisième région n'est liée à aucun fichier, c'est une région *anonyme* : il s'agit très certainement du *tas*, i.e. de la partie de l'espace utilisateur utilisée par les fameuses fonctions `malloc()`/`free()`. Les autres régions permettent de mapper le code ou les données de diverses bibliothèques partagées ou sont d'autres régions *anonymes* (sections `.bss` des exécutables mappés, pile des threads utilisateur du processus, etc.).

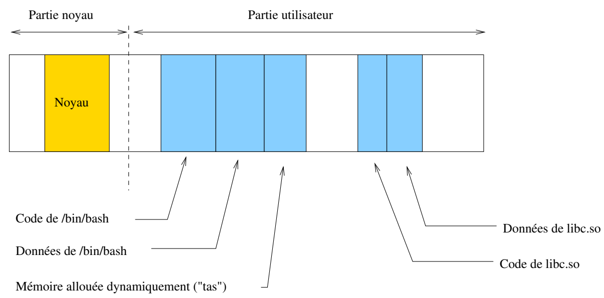

FIG. 2 – Représentation de l'espace d'adressage d'un processus `bash` avec quelques-unes de ses régions virtuelles

1.2 Fonctionnement de la projection en mémoire

Pour permettre le fonctionnement des régions virtuelles, le système d'exploitation utilise l'exception de "défaut de page". Celle-ci est levée par la MMU lorsqu'une traduction adresse virtuelle → physique n'est pas renseignée dans les tables de traduction (voir l'article 4).

Lors de la création d'une région virtuelle, le système d'exploitation n'associe aucune page physique à l'espace virtuel couvert par la région. Les tables couvrant l'espace de la région virtuelle ne contiennent donc pas de traduction adresse virtuelle → physique. Lorsque du code tentera d'accéder à une page de la région virtuelle, une exception de défaut de page sera levée. Celle-ci sera traitée par le système d'exploitation qui allouera une page physique, la mappera en mémoire virtuelle à l'adresse qui convient, remplira cette page avec les données correspondant à la ressource mappée, puis relancera l'application.

Ainsi, les pages physiques sont allouées et mappées à la demande au fil des défauts de page, on parle de *demand paging*.

1.2.1 Cas du mapping d'un fichier

Par exemple, lorsque la région virtuelle mappe un fichier et qu'un défaut de page intervient dans cette région, le système d'exploitation alloue une page physique, la mappe en mémoire virtuelle au bon endroit et y copie la portion du fichier correspondante. Une fois le défaut de page traité, l'application utilisateur poursuivra son exécution, avec l'impression d'accéder en mémoire au contenu du fichier.

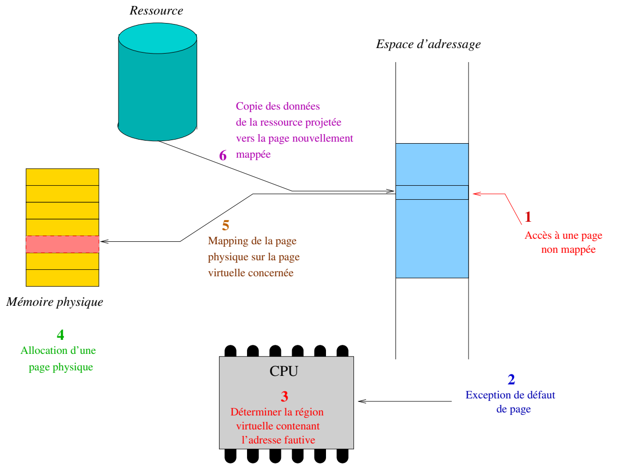

FIG. 3 – Traitement d'un *défaut de page* intervenant dans une région virtuelle projetant une ressource de type *fichier*.

C'est de cette façon que les programmes utilisateur et les bibliothèques dynamiques sont chargés sous Unix : ils sont simplement mappés en mémoire virtuelle. L'autre utilisation classique des mappings de fichiers est le partage de mémoire entre processus.

1.2.2 Cas d'un mapping anonyme

Lorsque la région virtuelle est de type *anonyme*, le principe est le même, sauf qu'au lieu d'initialiser la nouvelle page avec le contenu d'une portion d'un fichier, le système d'exploitation l'initialise à zéro avant de relancer l'application utilisateur. Celle-ci pourra alors utiliser cette zone de mémoire comme elle le souhaite.

Les mappings anonymes les plus courants sont les piles utilisateur des threads utilisateur et les zones de données utilisées par le *tas*, i.e. par les fonctions `malloc()`/`free()` (allocation/libération de mémoire pour l'application utilisateur) de la bibliothèque C. Le mapping de certains "fichiers" spéciaux peut également être strictement équivalent à un mapping anonyme : sous Unix, c'est le cas lorsqu'on mappe les périphériques `/dev/null` et `/dev/zero`.

1.3 Mapping de ressource et notion de *backing store*

Avec le *demand paging*, la mémoire physique sert de *cache* aux différentes ressources mappées. En effet, accéder à la mémoire virtuelle revient à accéder à une ressource, les données étant disponibles en mémoire physique pendant un petit moment. Comme la ressource mappée est en général initialement stockée sur disque, on obtient bien l'effet cache[1] classique : tant que la donnée est présente en mémoire physique, on y gagne en temps d'accès puisqu'on n'a pas à faire d'accès disque, et sinon on fait l'accès complet au disque mais le contenu reste accessible en mémoire physique un petit moment.

Comme pour les caches normaux, ce dispositif doit pouvoir fonctionner dans l'autre sens. On doit ainsi être capable de libérer les pages physiques les moins utilisées pour les utiliser au profit d'autres régions virtuelles plus souvent demandées. Dans ce cas, il faut être capable de libérer la mémoire physique de ces pages peu utilisées, mais sans perdre leur contenu. On doit pour cela les stocker en lieu sûr jusqu'à ce qu'elles soient de nouveau demandées. Ce "lieu sûr" est en général une partie d'un disque dur (local ou distant), il porte le nom de *backing store* (sauvegarde).

Dans le cas du mapping (de type "shared", nous en reparlerons plus loin) d'un fichier, le *backing store* est... le fichier. Cela signifie que ce fichier reflétera les modifications qu'on a effectuées en mémoire virtuelle. Précisons tout de même que chaque octet écrit en mémoire virtuelle ne sera pas immédiatement reporté sur disque, ce qui supprimerait l'effet cache du mapping de fichier. Les modifications seront reportées sur disque seulement quand le système d'exploitation ressentira le besoin de libérer les pages physiques mappant le fichier, ou lorsque l'application le demandera (appel système `msync()` sous Unix).

Dans le cas d'un mapping anonyme, le *backing store* est la zone d'échange (*swap*) si le système d'exploitation en offre un. Les mêmes remarques que pour le mapping de fichier s'appliquent.

Nous arrêtons ici ce tour d'horizon concernant le *backing store*. Dans SOS, nous n'avons pas prévu de nous intéresser à cette notion de *backing store* en général, ou au *swap* en particulier. Peut-être y reviendrons-nous quand même dans un ultime article...

1.4 Appels système Unix classiques

Afin de comprendre l'interaction de la mémoire virtuelle avec le reste du système, nous vous proposons d'étudier le fonctionnement de quelques appels système Unix importants.

1.4.1 Appels de la famille `mmap()`

Sous Unix, un des premiers appels système qui permet d'agir directement sur la mémoire virtuelle est `mmap()` ainsi que ses confrères `mremap()`, `mprotect()`, `munmap()` ou `msync()`. Ces fonctions permettent le mapping de fichiers mais servent aussi de base au partage de mémoire entre processus père/fils, i.e. liés entre eux par une série d'appels à `fork()` (voir plus loin). Les versions plus récentes d'Unix ont défini d'autres interfaces de programmation facilitant le partage de mémoire entre processus sans lien de parenté (IPC System V, notamment les fonctions de la famille `shmget()`) : dans le noyau, ces fonctions sont implémentées de façon très analogue à `mmap()`.

L'appel système principal qui nous intéresse ici, `mmap()`, permet de mapper une nouvelle région virtuelle dans l'espace d'adressage du processus courant. Il est possible soit de mapper un fichier, auquel cas la région virtuelle sera sa représentation en mémoire, ou bien de mapper de la mémoire anonyme (drapeau `MAP_ANONYMOUS`), auquel cas elle sera initialement à zéro.

Les autres appels système de la même famille permettent de démapper les régions d'une partie de l'espace virtuel utilisateur (`munmap`), de modifier les droits d'accès associés aux régions d'une partie de l'espace virtuel utilisateur (`mprotect`), de modifier la taille d'une région virtuelle (`mremap`) ou de synchroniser le contenu d'une région virtuelle avec son *backing store* (`msync`).

L'extrait de code source suivant montre l'utilisation de `mmap` et de `munmap` pour mapper en mémoire les 4 premiers kilo-octets du fichier `/etc/passwd`, puis utilise cette projection en mémoire pour afficher les 4 premiers caractères du fichier :

```
int fd = open("/etc/passwd", O_RDONLY);
void *ptr = mmap(NULL, 4096, PROT_READ, MAP_PRIVATE, fd, 0);
char *str = (char *) ptr;

printf(``%c %c %c %c\n'', str[0], str[1], str[2], str[3]);

munmap(ptr, 4096);
```

Le noyau du système d'exploitation doit donc permettre la création, la suppression et le redimensionnement de régions virtuelles. D'autre part, il doit associer à chaque région virtuelle le fichier sous-jacent ou son caractère *anonyme*. Lorsqu'un défaut de page intervient, le système d'exploitation doit :

1. déterminer la région virtuelle où le défaut de page a eu lieu. S'il a eu lieu en dehors des régions virtuelles du processus fautif, alors soit on le signale au processus (le fameux signal `SIGSEGV`, *aka* "Segmentation Fault" sous Unix), soit on termine le thread fautif du processus (cas de SOS pour l'instant) ;
2. allouer une page physique ;
3. mapper la page physique à l'emplacement mémoire où la faute a eu lieu ;
4. initialiser le contenu de cette nouvelle page, soit avec une portion du fichier mappé par la région virtuelle concernée, soit avec des zéros dans le cas d'une région virtuelle anonyme ;
5. relancer l'exécution de l'application ayant provoqué le défaut de page.

1.4.2 `fork()` et COW

Principe et utilisation. Sous Unix, l'appel système permettant de créer un nouveau processus est l'appel système

[1] Attention cependant : bien qu'on parle d'"effet cache", ce dispositif de *demand paging* ne correspond pas forcément à ce qu'on appelle "cache disque" ou "block cache". Dans le cas Linux 2.4 par exemple, les deux aspects sont dissociés.

fork(). Le nouveau processus s'exécute dans un nouvel espace d'adressage, mais est une copie conforme de son père : il possède les mêmes régions virtuelles, les mêmes fichiers ouverts, les mêmes propriétés.

L'extrait de code source suivant montre l'utilisation de l'appel système fork(). Avant le fork, un seul processus exécute le code. Suite au fork, deux processus exécutent le code :

– le processus qui préexistait, le *parent*, poursuit son exécution dans le else avec pid qui vaut le PID (identifiant de processus) du *fils* ;
– le nouveau processus, le *fils*, commence son exécution dans le if(pid == 0). Il exécute pour l'instant la même application que son père mais en est totalement indépendant.

```
pid_t pid;

pid = fork();

if(pid == 0) {
  printf(``Je suis dans le fils\n'');
}
else {
  printf(``Je suis dans le parent, mon fils est %d\n'', pid)
}
```

L'appel système fork nécessite l'intervention du sous-système de gestion de mémoire virtuelle pour la création d'un nouvel espace d'adressage et la recopie des régions virtuelles.

Types de mappings et "Copie-avant-écriture". Dans les premiers Unix, "recopie des régions virtuelles" signifiait qu'on recopiait le contenu de toutes les pages de ces régions du processus parent vers le processus fils. Cela rendait la création de nouveaux processus lente et rendait Unix gourmand en mémoire. Les Unix "modernes" (i.e. de moins de 30 ans) favorisent au maximum le partage de mémoire physique entre processus quand c'est possible, c'est-à-dire sans remettre en cause le cloisonnement mémoire fiable entre ceux-ci.

Ainsi, aujourd'hui "recopie des régions virtuelles" signifie qu'il y a recopie des régions virtuelles, au sens où ce sont les structures de données de description des régions virtuelles qui sont dupliquées. Il n'y a donc pas recopie du contenu de ces régions virtuelles du processus parent vers le processus fils. En réalité, immédiatement après un fork(), le père et le fils se partagent leurs pages de mémoire : les pages virtuelles aux mêmes adresses virtuelles dans les deux espaces d'adressage pointent sur les mêmes pages physiques (figure 4).

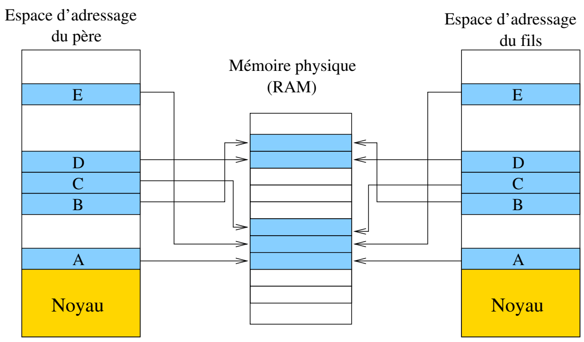

FIG. 4 – Partage de mémoire entre le processus père et le processus fils suite à un fork()

Pour les régions en lecture seule, ce partage ne pose pas de problème puisque la mémoire n'est pas modifiable.

En revanche, pour les régions en lecture/écriture, il peut y avoir problème : il n'est en effet pas toujours souhaitable que les deux processus partagent les modifications qu'ils apportent tous deux à la mémoire après le fork(). Par exemple, quand le serveur web apache lance un processus fils (appel à fork()) pour servir des requêtes HTTP, il ne faut pas que tout ce que fait le processus fils modifie les données ou la pile du père. Cela serait l'exact opposé de la caractéristique principale des processus : leur cloisonnement en mémoire.

On distingue ainsi deux types de régions virtuelles : les régions dites ***privées*** et les régions dites ***partagées***. Les régions *privées*, mappées à l'aide du drapeau MAP_PRIVATE de mmap(), sont gérées de manière à ce que le fils et le père ne voient que leurs propres modifications et jamais les modifications que l'autre processus apporte au contenu de la région. À l'inverse, les régions *partagées*, mappées à l'aide du drapeau MAP_SHARED, sont gérées de manière à ce que le fils et le père voient mutuellement les modifications qu'ils réalisent sur la région.

Pour les régions en lecture/écriture et de type *partagé*, la gestion est relativement simple : le père et le fils continuent à partager les pages physiques contenant les données. Non seulement le fait de partager ces pages au lieu de les copier a été un gain de temps considérable. Mais en plus ces données ne sont présentes qu'une seule fois en mémoire physique, quel que soit le nombre de processus fils créés, d'où une économie importante de mémoire physique.

Pour les régions de type *privé*, la gestion est un peu plus fine. Tant que les accès se font en lecture, il est possible de partager les pages physiques entre le père et le fils. Dès lors que l'un d'entre eux tente de faire un accès en écriture, il faut réaliser une copie de la page concernée dans une nouvelle page et mapper cette nouvelle page en lecture/écriture à la place de l'ancienne dans l'espace d'adressage du processus écrivain. Ce dernier possède donc maintenant sa propre copie de la page, qu'il peut dès lors modifier comme bon lui semble. Ni les autres processus, ni le fichier initial ne seront plus affectés par ces modifications : cette page est donc autonome et ne mappe plus le fichier initial, il s'agit en fait d'une forme particulière de mapping anonyme.

Ce mécanisme est appelé *Copy-on-write* (*COW*), "copie avant écriture" en français. Sa mise en oeuvre est assez simple : lors d'un fork(), les pages virtuelles correspondant à une région virtuelle en lecture/écriture de type *privé* sont mappées en lecture seule à la fois chez le père et le fils. Ainsi, lors de la première écriture, soit par le père, soit par le fils, une exception de défaut de page sera levée. Le système d'exploitation constatera qu'il s'agit d'une violation de protection en écriture alors que la région virtuelle correspondante est de type lecture/écriture : ceci est le déclencheur du mécanisme de *Copy-on-write*.

Dans la figure 4, on peut supposer que la région virtuelle composée des pages B, C et D est de type *privé* et en lecture/écriture. Cette figure représente l'état de partage des pages tant qu'aucune écriture n'a été réalisée par aucun des processus. Si le processus fils tente une écriture sur la page D, le système d'exploitation déclenchera le mécanisme de *Copy-on-write*, et on obtiendra la situation de la figure 6 : une page physique verte a été allouée, le contenu de la page D y a été copié et la page physique verte est mappée à la place

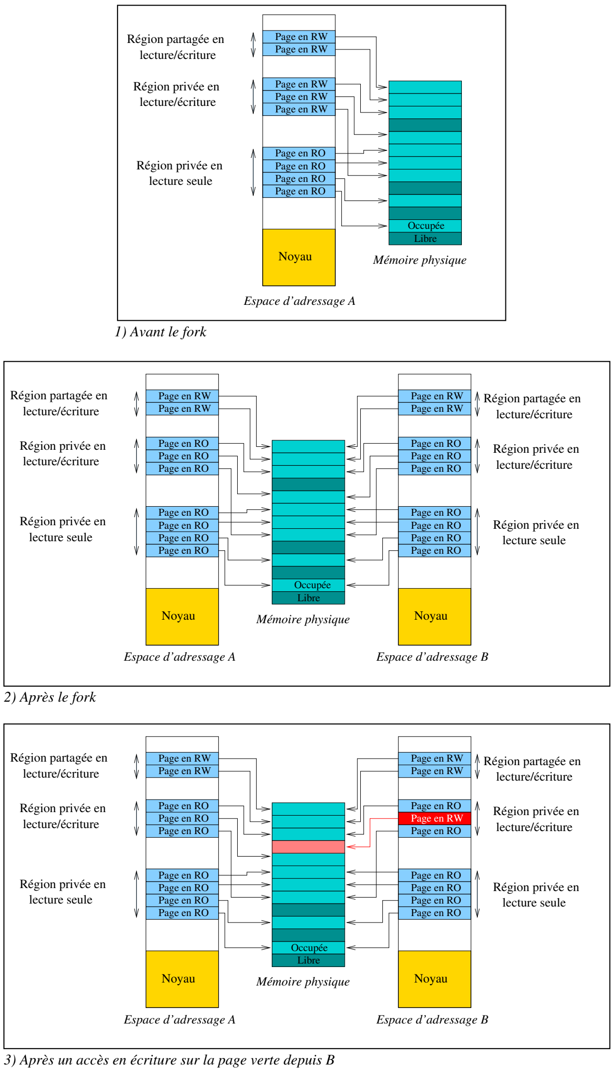

FIG. 5 – Mécanisme "COW". **Étape 1**, un processus s'exécute dans un espace d'adressage A, et exécute un `fork`. **Étape 2**, après le `fork`, un nouveau processus similaire à A s'exécute dans un espace d'adressage B. Les pages de la région privée en lecture/écriture ont été passées en lecture seule. **Étape 3**, un accès depuis l'espace d'adressage B sur la page verte a engendré un défaut de page, déclenchant le mécanisme de *Copy-on-write*. Une nouvelle page physique rouge a été allouée et mappée en lecture/écriture et contiendra les modifications effectuées par B.

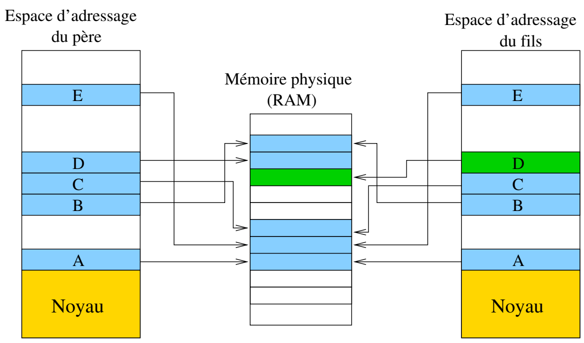

FIG. 6 – Suite à une écriture du processus fils sur la page D, le système d'exploitation a déclenché un *Copy-on-write* sur cette page. Le processus fils dispose maintenant de sa propre copie de D qu'il peut modifier.

	Type privé	Type partagé
Lecture seule	Lecture = OK Écriture = Segfault	Lecture = OK Écriture = Segfault
Lecture/écriture	Lecture = OK Écriture = COW puis OK	Lecture = OK Écriture = OK

TAB. 1 – Comportement de la mémoire virtuelle en fonction du type de la région virtuelle et de ses droits d'accès

de D dans le processus fils : il pourra la modifier sans affecter ni le père, ni le fichier sous-jacent (cette page est devenue un mapping anonyme). Les deux processus n'ont plus exactement la même représentation de la mémoire : il y a eu *différenciation*. Le processus père dispose maintenant d'une page physique ayant le contenu de D qui n'est utilisée que par lui : il peut donc y accéder en lecture/écriture sans géner son processus fils. Il ne sera donc pas nécessaire d'effectuer un *Copy-on-write* lors de l'accès en écriture à cette page par le processus père.

Ces mécanismes de partage de page et de différenciation par *Copy-on-write* ont plusieurs avantages :

– le partage de pages permet d'éviter de coûteuses (en temps et en mémoire) recopies mémoire lors d'un `fork`;
– le *Copy-on-write* permet de reporter le plus tard possible les copies mémoire qui sont réellement nécessaires.

Le tableau 1 résume le comportement du système de gestion de mémoire virtuelle vis à vis des différents types de régions virtuelles et de leurs droits d'accès.

1.4.3 Appel `exec`

Il est impossible de concevoir un système généraliste capable de lancer des programmes différents si tous les processus créés sont tous identiques à un père commun. `fork()`, bien qu'élégant, ne suffit donc pas.

Pour lancer de nouveaux programmes sous Unix, l'appel système `fork()` est utilisé de concert avec un autre appel système, l'appel système `exec()`. Ce dernier permet de remplacer l'application en exécution dans le processus courant par une autre application en la chargeant à partir de son fichier binaire, stocké dans un système de fichiers.

L'extrait de code source suivant montre schématiquement comment un *shell* lance l'application `ls` lorsque cela lui est demandé. Il commence par créer un nouveau processus en utilisant `fork()` puis dans ce nouveau processus, exécute l'application `ls`. Au niveau du père, il utilise l'appel système `wait()` pour attendre la terminaison du fils.

```
pid_t pid;

pid = fork();

if(pid == 0) {
  /* Lancer l'exécution de 'ls' */
  execl(``/bin/ls'', NULL);
}
else {
  /* Attendre la terminaison du fils */
  int status;
  wait(&status);
}
```

Dans les Unix "modernes", `exec()` est réalisé en utilisant uniquement les mécanismes de mémoire virtuelle. Ainsi, on n'écrit pas directement le code et les données du nouveau programme en mémoire physique lors de son chargement. On se contente de mapper le fichier exécutable en mémoire virtuelle au bon endroit, et on profite ainsi des avantages du *demand paging*.

L'appel système `exec()` consiste donc à supprimer toutes les régions virtuelles de l'espace d'adressage courant, puis à mapper l'exécutable dans de nouvelles régions virtuelles à partir de la carte mémoire du fichier binaire de la nouvelle application à exécuter.

Pour cela, il utilise l'*entête de programme* (*Program Header*) du fichier binaire, au format ELF (voir [7]). Cet entête de programme est déjà utilisé par SOS depuis l'article 7, afin de charger les applications utilisateur. Il indique quelles portions du fichier doivent être mappées en mémoire et avec quels droits d'accès.

Ainsi, pour le programme `/bin/ls` :

```
$ readelf -l /bin/ls
[...]
En-têtes de programme:
  Type           Décalage Adr. vir.  Adr.phys.  T.Fich. T.Mém.  Fan Alignement
  PHDR           0x000034 0x08048034 0x08048034 0x00100 0x00100 R E 0x4
  INTERP         0x000134 0x08048134 0x08048134 0x00013 0x00013 R   0x1
      [Réquisition de l'interpréteur de programme: /lib/ld-linux.so.2]
  LOAD           0x000000 0x08048000 0x08048000 0x11d08 0x11d08 R E 0x1000
  LOAD           0x012000 0x0805a000 0x0805a000 0x003f4 0x007b0 RW  0x1000
  DYNAMIC        0x012184 0x0805a184 0x0805a184 0x000d8 0x000d8 RW  0x4
  NOTE           0x000148 0x08048148 0x08048148 0x00020 0x00020 R   0x4
  GNU_EH_FRAME   0x011cdc 0x08059cdc 0x08059cdc 0x0002c 0x0002c R   0x4
  STACK          0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0x4
[...]
```

il y a deux portions du fichier binaire à charger en mémoire (i.e. de type `LOAD`), l'un débutant à l'offset 0 du fichier, à charger à l'adresse `0x8048000` en lecture/exécution (`RE`) et l'autre situé à l'offset `0x12000` du fichier, à charger à l'adresse `0x805a000` en lecture/écriture (`RW`).

1.5 Régions spécifiques

Certaines régions spécifiques ont un comportement légèrement différent des autres : le tas et la pile.

1.5.1 Le tas

Le *tas* est une région particulière réservée pour les allocations dynamiques de mémoire réalisées par l'application utilisateur. Les fonctions `malloc()` et `free()` de la bibliothèque C utilisateur s'occupent de gérer cette région pour satisfaire les requêtes d'allocation/libération. Cette région est en réalité un mapping anonyme privé comme un autre, mais le noyau doit permettre à la bibliothèque C de l'identifier très précisément, ou doit s'occuper de le gérer. Dans les deux cas, c'est une fonction particulière, `brk()`, qui permet de modifier sa taille en fonction des besoins. Dans le noyau Linux, `brk()` est en fait un appel système car c'est le noyau qui s'occupe d'identifier cette région particulière et de la gérer.

L'implémentation de `malloc()` de la bibliothèque C peut donc utiliser `brk()` pour satisfaire les allocations, mais il peut tout aussi bien utiliser des `mmap()` classiques. Par exemple, la `GNU libc` utilise les deux options et elle décide laquelle choisir en fonction de la taille des allocations demandées.

1.5.2 La pile

La pile utilisée au niveau utilisateur est également une région virtuelle correspondant à un mapping anonyme privé. Dans certains OS tels que Linux, cette région est cependant un peu particulière puisqu'elle peut s'étendre dynamiquement. Sur architecture Intel, la pile s'agrandit vers les adresses basses, donc dans ces OS la région la contenant s'étend dynamiquement vers le bas. Ces régions sont marquées du drapeau `MAP_GROWSDOWN` dans Linux par exemple. Lorsqu'un défaut de page survient en dehors d'une région, mais à proximité du bas de la région de la pile, alors cette dernière est automatiquement étendue. L'application peut ainsi utiliser de grandes quantités de mémoire sur la pile.

2 Implémentation

L'implémentation de la mémoire virtuelle que nous vous proposons ce mois-ci dans SOS est un peu plus limitée que la description que nous en avons faite en partie 1, sur plusieurs aspects :

- SOS ne disposant pas de système de fichiers et de pilotes de périphériques permettant le stockage de données, il n'est pas possible pour l'instant de projeter de *vrais* fichiers en mémoire. Nous proposons une implémentation dans laquelle les ressources "projetables" que nous implémenterons simuleront des fichiers en utilisant de la mémoire noyau ;
- la pile utilisateur ne sera pas gérée via un mécanisme d'extension similaire à `MAP_GROWSDOWN`. La pile utilisateur aura une taille, en mémoire virtuelle, fixée à la création. Si un redimensionnement est souhaité, alors il devra être réalisé explicitement par un appel à une fonction du sous-système de gestion de mémoire virtuelle (`sos_umem_vmm_resize`);

Comme d'habitude, pour suivre les explications qui suivent, il est fortement conseillé d'avoir le code sous les yeux. Ce mois-ci, le code n'est pas présent sur le CDROM : reportez-vous à la page "Téléchargements" du site de SOS (`http://sos.enix.org`).

2.1 Aperçu général

L'implémentation de la gestion de la mémoire virtuelle dans SOS est découpée en différents sous-systèmes qui interagissent (figure 7) :

- Un **sous-système de gestion des régions virtuelles**, chargé de faire de l'arithmétique sur les intervalles pour gérer des régions virtuelles. Il permet de déterminer quelles sont les zones libres de l'espace d'adressage, de créer, de supprimer, de redimensionner les régions virtuelles ou de changer leurs droits d'accès. Son rôle est limité à ces manipulations d'intervalles ;
- Un **sous-système de manipulation des tables de pages**, implémenté principalement au cours des articles précédents (sous-système `paging`) permet de mapper, de démapper des pages et de changer leurs droits d'accès ;
- Un **sous-système de gestion des défauts de pages**, qui effectue les opérations nécessaires pour résoudre un défaut de page signalé par le processeur. Il est chargé de réaliser le *Copy-on-write* lorsque c'est nécessaire, et de s'adresser en cas de besoin au **pilote de ressource mappée** sous-jacente pour effectuer du *demand paging* ;
- Un ou plusieurs **pilotes de ressource mappée** qui définissent le comportement de la ressource mappée vis-à-vis des évènements signalés par le système de ges-

tion de mémoire virtuelle au travers de différents *call-back*. En particulier, un *call-back* spécifique permet d'effectuer le *demand-paging* sur demande du **sous-système de gestion des défauts de pages**. L'approche choisie est de type orientée-objet. Ainsi, ces pilotes définissent des structures qui contiennent des *méthodes* : les *call-backs*.

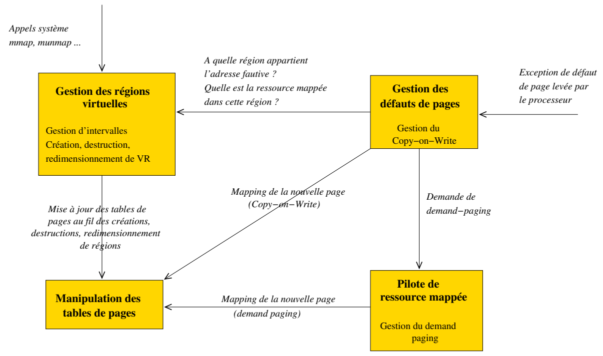

FIG. 7 – Aperçu général des différents sous-systèmes et de leurs interactions.

2.2 Mécanismes bas niveau : manipulation des tables de pages

Afin de permettre la copie d'espace d'adressage pour le `fork` et le mécanisme de *Copy-on-write*, il est nécessaire d'ajouter quelques mécanismes bas-niveau concernant la gestion de la pagination.

2.2.1 Duplication d'un espace d'adressage

Tout d'abord, une nouvelle fonction `sos_mm_context_duplicate()` a été implémentée dans `hwcore/mm_context.c`. Elle permet de créer un nouvel espace d'adressage qui est une copie d'un espace d'adressage passé en paramètre. En premier lieu, un nouveau `mm_context` est alloué en utilisant `sos_mm_context_create()`. Cette dernière se charge de synchroniser la partie noyau du nouvel espace d'adressage en utilisant `sos_paging_copy_kernel_space()` (voir l'article 7). Ensuite, `sos_paging_copy_user_space()` est appelée pour copier la partie utilisateur de l'espace d'adressage original dans le nouvel espace d'adressage. Ainsi, au final, le nouvel espace d'adressage sera en tous points similaire à l'original.

La fonction `sos_paging_copy_user_space()` de `hwcore/paging.c` fonctionne en recopiant les tables de pages (*PT*) de l'espace d'adressage source vers l'espace d'adressage destination. Pour cela, pour chaque PT de la partie utilisateur, la fonction alloue une nouvelle page physique pour le nouveau PT, mappe le PT source et le PT destination temporairement en zone noyau, et recopie le PT source dans le PT destination. En même temps, elle incrémente le compteur de référence des pages physiques référencées par les différents PTs. De cette manière, une page physique mappée en zone utilisateur dans l'espace d'adressage original ayant un compteur de référence égal à 1, verra celui-ci passer à 2. En effet, suite à la copie, cette page est maintenant mappée par deux espaces d'adressage différents.

On peut noter qu'au niveau de la partie noyau (fonction `sos_paging_copy_kernel_space`), les PTs sont partagées entre les différents espaces d'adressage, alors qu'au niveau de la partie utilisateur, chaque espace d'adressage dispose de ces propres PTs. En effet, au niveau de la zone noyau, les PTs de tous les espaces d'adressage doivent toujours être identiques. En revanche, au niveau utilisateur, les PTs divergent de manière différente au fil des *Copy-on-write* (voir figure 8). Il serait possible de partager les PTs de la partie utilisateur entre un espace d'adressage fils et un espace d'adressage père, et de n'allouer les nouveaux PTs qu'à la demande, en utilisant du *Copy-on-write* sur les PTs. Pour des raisons de simplicité, nous avons choisi de ne pas implémenter ce mécanisme, le lecteur pourra s'y essayer à titre d'exercice.

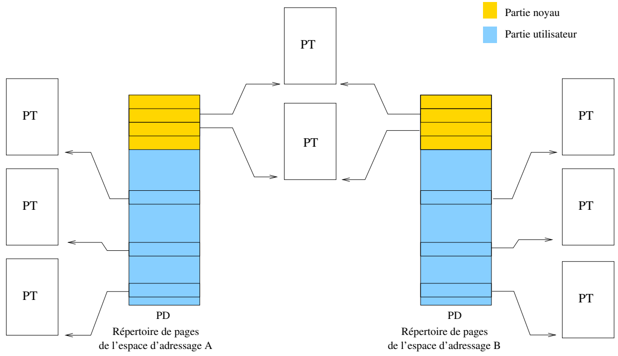

FIG. 8 – L'espace d'adressage B a été créé par duplication de l'espace d'adressage A. Ces deux espaces d'adressage partagent leurs PTs en ce qui concerne la zone noyau (jaune). En revanche, en ce qui concerne la zone utilisateur (bleue), ils disposent chacun de leurs propres PTs, qui sont initialement identiques.

2.2.2 Gestion du *Copy-on-write*

Le fichier `hwcore/paging.c` propose deux autres nouvelles fonctions. La première, `sos_paging_prepare_COW()`, prépare un ensemble de pages de l'espace d'adressage pour le *Copy-on-write*. Pour cela, il suffit de changer la protection des pages en "lecture seule" même si la région qui les couvre est de type lecture/écriture. Ainsi, tout accès en écriture sur ces pages déclenchera une exception de défaut de page, que SOS pourra résoudre en effectuant le *Copy-on-write*.

La seconde, `sos_paging_try_resolve_COW()`, permet d'effectuer le *Copy-on-write* proprement dit à partir de l'adresse fautive. Le gestionnaire de défaut de page utilise cette fonction lorsqu'un défaut de page a été signalé sur une page mappée en lecture seule dans une région en lecture/écriture. Cette fonction regarde tout d'abord le compteur de référence de la page physique concernée par le *Copy-on-write*. Si celui-ci vaut un, cela signifie que l'espace d'adressage courant est le seul à faire référence à cette page, et qu'on peut donc l'utiliser directement en la passant en lecture/écriture. Ce cas arrive lorsque tous les autres espaces d'adressage mappant cette page ont déjà effectué leur propre copie privée de cette page, et que l'espace d'adressage courant est donc le dernier à la mapper. Sinon, si ce compteur de référence est différent de un, alors une nouvelle page physique est allouée, le contenu de la page à *différencier* y est copié, puis la page physique est mappée en lecture/écriture à la place de la page fautive (voir figures 9, 10, 11).

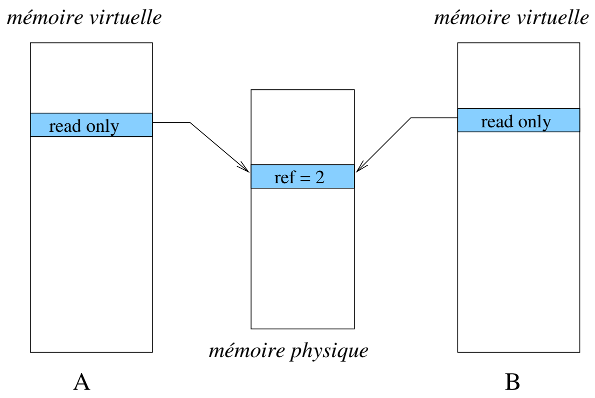

FIG. 9 – B est un espace d'adressage issu d'un `fork` de A, ils partagent une page dans une région en lecture/écriture. Dans A et B la page a été passée en lecture seule pour permettre le fonctionnement du *Copy-on-write*. La page physique bleue est donc référencée par deux pages virtuelles.

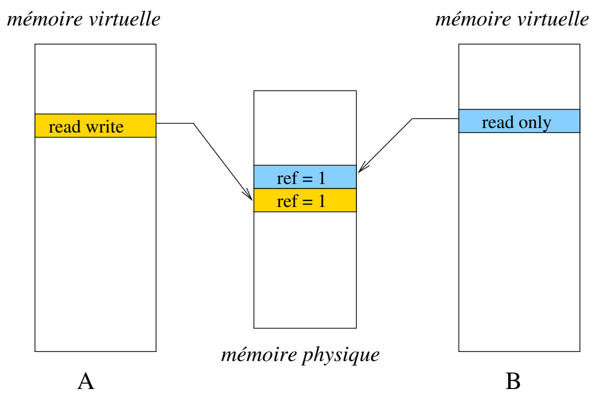

FIG. 10 – L'application fonctionnant dans A a tenté un accès en écriture sur la page. Un défaut de page est donc déclenché, le gestionnaire de cette exception s'aperçoit que la page est mappée en lecture seule dans une région en lecture/écriture et déclenche le mécanisme de *Copy-on-write*. Une nouvelle page physique (jaune) est allouée, le contenu de la page bleue y est copié, puis elle est mappée dans l'espace d'adressage A. La page physique bleue n'est plus référencée que par une page virtuelle.

On peut rappeler l'utilité de la macro `invlpg` utilisée après avoir passé la page en lecture/écriture lorsque le compteur de référence de la page est à 1. Elle permet de forcer le cache de traduction d'adresse (TLB) à prendre en compte les modifications que l'on vient d'apporter aux tables de pages (voir article 4 pour plus de détails sur le fonctionnement du TLB).

2.3 Espaces d'adressage et régions virtuelles

L'implémentation des mécanismes de haut-niveau, notamment la gestion des régions virtuelles, est présente dans le fichier `sos/umem_vmm.c` et son entête `sos/umem_vmm.h`.

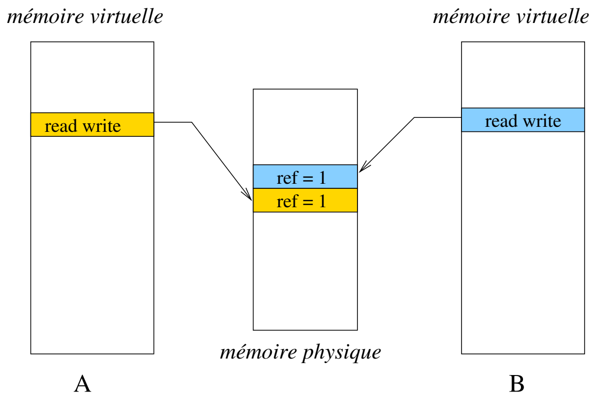

FIG. 11 – L'application fonctionnant dans B a tenté un accès en écriture sur la page bleue. Le même mécanisme que précédemment se met en marche. Cependant, ici, la page physique bleue a déjà un compteur de référence à 1 : elle n'est donc utilisée que par B. Il suffit donc de la passer en lecture/écriture pour relancer l'application.

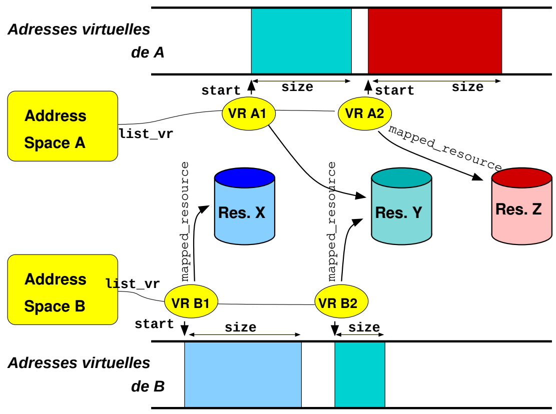

FIG. 12 – Éléments principaux de la gestion de la mémoire virtuelle dans SOS. Exemple avec 2 processus A et B mappant chacun deux ressources au travers de 2 régions virtuelles, dont une est un partage de la ressource Y.

2.3.1 Structures de données

La figure 13 résume les relations entre les différentes structures de données que nous voyons ci-dessous.

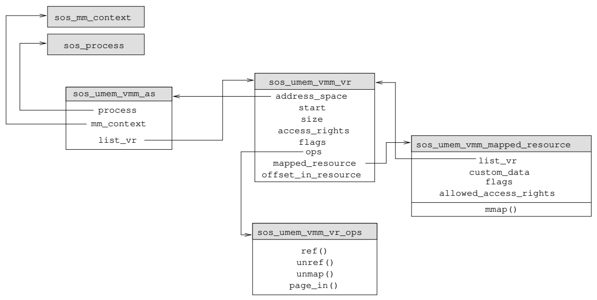

FIG. 13 – Organisation des structures de données liées à la gestion de la mémoire virtuelle.

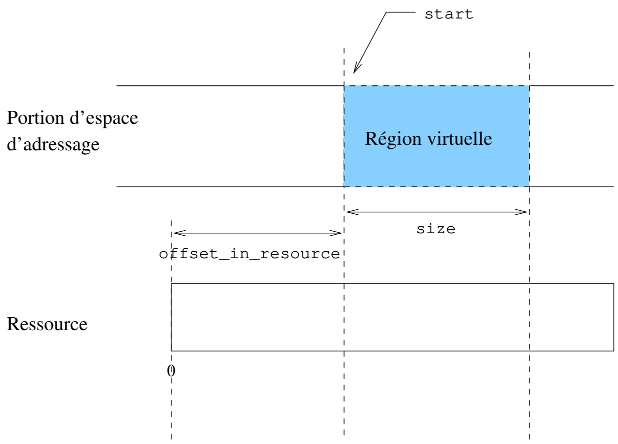

FIG. 14 – Une ressource est projetée en mémoire au travers d'une région virtuelle. Seule la partie comprise dans l'intervalle *[offset_in_resource ; offset_in_resource + size]* est projetée en mémoire.

Espace d'adressage. La structure `sos_umem_vmm_as` est la représentation de haut-niveau d'un espace d'adressage, c'est-à-dire l'association entre un contexte mémoire `sos_mm_context` et une liste de régions virtuelles. Cette structure comporte également divers champs à usage statistique.

Pour chaque espace d'adressage, la liste des régions virtuelles est classée dans l'ordre des adresses de début des régions croissantes, sans recouvrement possible entre les régions.

Région virtuelle. La structure `sos_umem_vmm_vr` est la représentation d'une région virtuelle. Celle-ci est caractérisée par une adresse de début `start`, une taille `size`, des droits d'accès `access_rights`, des drapeaux `flags` et appartient à un espace d'adressage. Le rôle d'une région virtuelle est de représenter en mémoire le contenu d'une *ressource*, la `mapped_resource` (voir 2.3.1). Chaque région virtuelle est sur 2 listes :

– la liste des régions virtuelles de l'espace d'adressage auquel elle appartient,
– la liste des régions virtuelles qui mappent la ressource sous-jacente. Plusieurs régions virtuelles d'espaces d'adressage différents peuvent mapper la même ressource.

Le champ `offset_in_resource` de la structure `sos_umem_vmm_vr` a une sémantique différente en fonction du type de région.

Si la région n'est pas une région *anonyme*, il désigne l'offset dans la ressource à partir duquel son contenu est projeté en mémoire (voir figure 14). Par exemple, si une région virtuelle projette une ressource avec un `offset_in_resource` de 4096, cela signifie que le premier octet de la région correspondra au 4096ème octet de la ressource. Pour un exemple d'utilisation, voir l'implémentation du pilote de chargement des applications binaires, en partie 2.6.

Si la région est anonyme, le champ `offset_in_resource` est également renseigné même si, par définition, il n'y a pas de réel fichier sous-jacent. Son rôle est de situer chaque page mappée dans le mapping anonyme par rapport à la ressource "virtuelle" correspondant au mapping anonyme. Le fait de savoir situer chaque page dans cette ressource virtuelle va ainsi permettre de réaliser du partage de pages entre différents processus lorsque le mapping est de type partagé, sans se tromper dans le cas où la région est agrandie, change d'adresse, etc. La valeur de ce champ est arbitraire mais il faut se fixer une convention pour éviter les incohérences. Dans SOS, pour éviter des débordements en arithmétique entière en cas d'élargissement "par le bas" de la région, on initialise ce champ à l'adresse de début de la région virtuelle mappée.

Chaque région virtuelle est également associée à une série d'opérations ou *callbacks*, que le système de gestion de mémoire virtuelle appelera lorsque c'est nécessaire. Ces opérations sont définies par la structure `sos_umem_vmm_vr_ops` et doivent être implémentées par le pilote en charge de la ressource mappée dans la région virtuelle (voir 2.5 et 2.6). Les opérations sont au nombre de quatre :

– **ref** est appelée lorsque la région virtuelle est ajoutée à un espace d'adressage ;
– **unref** lorsqu'elle est retirée d'un espace d'adressage ;
– **unmap** est appelée lorsque que tout ou partie de la région est supprimé ;
– **page_in** est l'opération la plus importante. Elle est appelée lorsqu'un défaut de page a été levé et que le système de gestion de mémoire virtuelle demande à la ressource de mapper les données associées à la région concernée (voir 2.4).

C'est donc toujours la ressource qui s'occupe de mapper les données. Le sous-système `umem_vmm` ne fait que gérer les intervalles et signaler aux ressources qu'elles doivent mapper leurs données.

Ressource mappée. Une ressource mappée est définie par une structure `sos_umem_vmm_mapped_resource`. La séparation entre région virtuelle et ressource mappée permet de bien distinguer les deux concepts. La ressource est un objet du système (dans Unix : un fichier ouvert) alors que la région virtuelle en est une représentation dans un espace d'adressage. Une même ressource peut être mappée dans différents espaces d'adressage, avec éventuellement différents droits d'accès, grâce à plusieurs régions virtuelles.

La structure `sos_umem_vmm_mapped_resource` contient notamment la liste des régions virtuelles qui mappent cette ressource ainsi qu'un pointeur vers la méthode "`mmap`" appelée lorsque la ressource est mappée en mémoire. Cette méthode sera principalement utilisée pour indiquer au système de gestion de mémoire virtuelle quelles sont les opérations *ref*, *unref*, *page_in* à utiliser pour la région.

2.3.2 Manipulation des espaces d'adressage

La fonction `sos_umem_vmm_create_empty_as()` créé un nouvel espace d'adressage vide (sans régions virtuelles), tandis que `sos_umem_vmm_delete_as` supprime un espace d'adressage ainsi que les régions virtuelles et le contexte mémoire qu'il contient. La fonction `sos_umem_vmm_duplicate_as()` est plus intéressante : elle permet de dupliquer un espace d'adressage, opération nécessaire lors d'un `fork()`. Cela consiste à allouer un nouvel espace d'adressage, puis pour chaque région virtuelle du parent, à créer une région virtuelle correspondante dans le nouvel espace d'adressage. Dans le même temps, si la région virtuelle n'est pas du type partagé, alors elle est préparée pour fonctionner en *Copy-on-write* par l'appel à `sos_paging_prepare_COW()`. Une fois que toutes les régions virtuelles sont présentes dans le nouvel espace d'adressage, les tables de pages de ce dernier sont copiées depuis le parent grâce à `sos_mm_context_duplicate()`.

2.3.3 Manipulation des régions virtuelles

SOS propose une API inspirée de celle du `mmap()` Unix pour la manipulation des régions virtuelles.

Mapper une ressource. La fonction `sos_umem_vmm_map()` (analogue au `mmap()` d'Unix) permet de mapper une ressource en mémoire en créant une nouvelle région virtuelle. Elle prend un certain nombre de paramètres définissant les caractéristiques de la région à créer : espace d'adressage, adresse de début, taille, droits d'accès, ressource mappée, etc..

Après quelques vérifications, la fonction commence par allouer la nouvelle structure `sos_umem_vmm_vr` puis détermine l'adresse de début de la région. Si l'adresse est imposée (drapeau `SOS_VR_MAP_FIXED`), alors les éventuelles régions occupant le même espace sont supprimées. Sinon, la fonction `find_fist_free_interval()` essaie de trouver un emplacement libre suffisamment grand à proximité de l'adresse `uaddr` fournie.

Pour résumer, il y a cinq cas pour déterminer l'adresse de début d'une région virtuelle :

- **Cas 1** : drapeau `SOS_VR_MAP_FIXED` positionné, adresse positionnée dans `uaddr` et emplacement dans l'espace d'adressage disponible suffisamment grand à l'adresse `uaddr` → l'adresse `uaddr` est utilisée ;
- **Cas 2** : drapeau `SOS_VR_MAP_FIXED` positionné, adresse positionnée dans `uaddr` et emplacement dans l'espace d'adressage non disponible → les régions virtuelles occupant l'espace sont supprimées, et l'adresse `uaddr` est utilisée ;
- **Cas 3** : drapeau `SOS_VR_MAP_FIXED` non positionné, adresse positionnée dans `uaddr` et emplacement dans l'espace d'adressage disponible suffisamment grand à l'adresse `uaddr` → l'adresse `uaddr` est utilisée ;
- **Cas 4** : drapeau `SOS_VR_MAP_FIXED` non positionné, adresse positionnée dans `uaddr` et emplacement dans l'espace d'adressage non disponible → recherche d'un autre emplacement libre suffisamment grand pour la région ;
- **Cas 5** : drapeau `SOS_VR_MAP_FIXED` non positionné, pas d'adresse spécifiée dans `uaddr` → sélection d'un emplacement libre pour la région.

Une fois l'emplacement de la région virtuelle déterminé, la fonction regarde s'il est possible de combiner la région en création avec la région précédente ou la région suivante, relativement à l'ordre des adresses virtuelles. Pour que ce soit le cas, il faut que les régions soient de même type, que les adresses correspondent et que la ressource mappée soit identique. S'il est possible de combiner la région en création avec une région pré-existante, alors cette dernière est simplement agrandie en conséquence. Sinon, la structure `sos_umem_vmm_vr` précédemment allouée est initialisée puis ajoutée à la liste des régions virtuelles de l'espace d'adressage.

Démapper une ressource. La fonction `sos_umem_vmm_unmap()` (analogue au `munmap()` d'Unix) permet de démapper une partie d'un espace d'adressage, en spécifiant une adresse de début et une taille. Cette portion de l'espace d'adressage peut couvrir une ou plusieurs régions, en totalité ou en partie. Cette souplesse de fonctionnement rend l'implémentation assez délicate : elle nécessite un peu de gymnastique d'intervalles pour manipuler les régions.

La fonction s'articule autour d'une boucle `while` qui balaye la liste des régions de l'espace d'adressage dans le sens des adresses croissantes et qui s'exécutera tant qu'il restera des régions à supprimer ou à découper dans l'intervalle donné par l'appelant. À chaque itération, il y a quatre cas possibles :

- **Cas 1** : la région en cours de traitement est entièrement contenue dans l'intervalle spécifié, elle est donc totalement supprimée (voir figure 15). Le traitement se poursuit à la prochaine région de l'espace d'adressage ;
- **Cas 2** : l'intervalle spécifié est intégralement inclus dans la région en cours de traitement. Cette dernière doit donc être coupée en deux parties : une première partie $[debut\ de\ la\ region\ ;\ adresse\ donnee\ par\ l'utilisateur]$ et une seconde partie $[adresse\ donnee\ par\ l'utilisateur\ +\ taille\ donnee\ par\ l'utilisateur\ ;\ fin\ de\ la\ region]$. Pour réaliser ceci, la région originelle, qui couvre maintenant la première partie, est rétrécie et une nouvelle région est allouée pour couvrir la deuxième partie (voir figure 16). Puisque l'ensemble de l'intervalle spécifié par l'appelant a été traité, la boucle `while` s'arrête là ;
- **Cas 3** : l'intervalle affecte uniquement le début de la région virtuelle en cours de traitement. L'adresse de début de celle-ci est donc décalée pour libérer la place et sa taille est réduite (voir figure 15). La boucle s'arrête ici, puisque si seul le début de la région virtuelle a été affecté, c'est que l'intervalle spécifié par l'appelant a été traité dans sa totalité ;
- **Cas 4** : l'intervalle affecte uniquement la fin de la région virtuelle en cours de traitement. La taille de la région est donc décrémentée pour libérer la place (voir figure 15). La prochaine itération de la boucle est réalisée avec la région virtuelle suivante dans la liste.

Une fois que l'espace à démapper n'est plus couvert par une région virtuelle, il est nécessaire de démapper les pages physiques présentes dans cette zone. En ef-

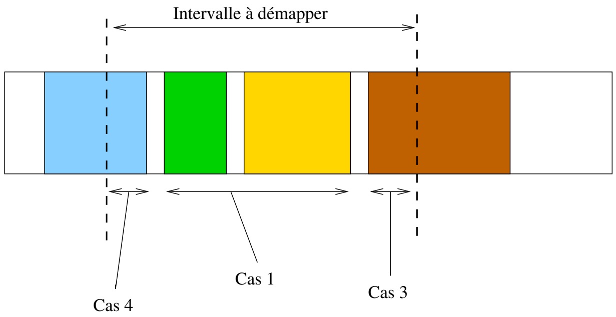

FIG. 15 – Illustration des cas 1, 3 et 4 rencontrés dans la fonction `sos_umem_vmm_unmap()`

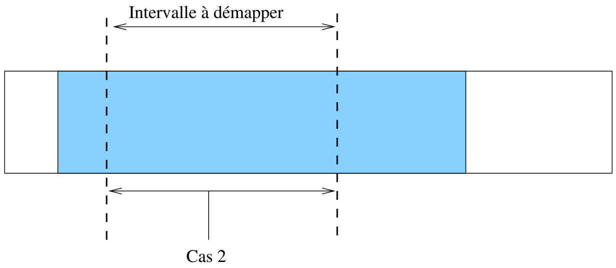

FIG. 16 – Illustration du cas 2 rencontré dans la fonction `sos_umem_vmm_unmap`

fet, dès lors que la région n'existe plus, il ne doit plus être possible d'accéder aux données précédemment mappées, et tout accès doit déclencher une exception de défaut de page. Pour démapper ces pages, la fonction `sos_paging_unmap_interval()` est utilisée. Un changement d'espace d'adressage a éventuellement lieu si la région virtuelle manipulée est située dans un espace d'adressage qui n'est pas l'espace d'adressage courant.

Modification des droits d'accès. La fonction `sos_umem_vmm_chprot()` (analogue au `mprotect()` d'Unix) permet de modifier les droits d'accès d'une portion de l'espace d'adressage. Cette portion peut couvrir seulement une partie d'une région ou bien plusieurs régions. Son implémentation est donc très similaire à celle de `sos_umem_vmm_unmap()` : articulée autour d'une boucle permettant de traiter toutes les régions virtuelles couvertes par l'intervalle d'adresses passé en paramètre, elle gère les différents cas de recouvrements.

Toutefois, contrairement à `sos_umem_vmm_unmap()` qui pouvait amener à la création d'**une** région virtuelle, `sos_umem_vmm_chprot()` peut amener à la création de **deux** régions virtuelles. En effet, quand l'intervalle spécifié par l'appelant se situe au milieu d'une région virtuelle, on obtient trois régions virtuelles. Il est donc nécessaire de créer deux nouvelles régions virtuelles. Ce cas **1** est illustré sur la figure 17 avec les autres cas.

Parallèlement aux manipulations de régions virtuelles, la fonction `sos_umem_vmm_chprot()` effectue les manipulations correspondantes sur les tables de pages. En effet, tout changement de droits d'accès sur une région virtuelle doit être répercuté sur les tables de pages : ce sont elles qui déterminent quels accès sont autorisés et quels accès entraîneront une exception de défaut de page. Pour la mise en place des nouveaux droits sur les tables de pages, deux cas se présentent :

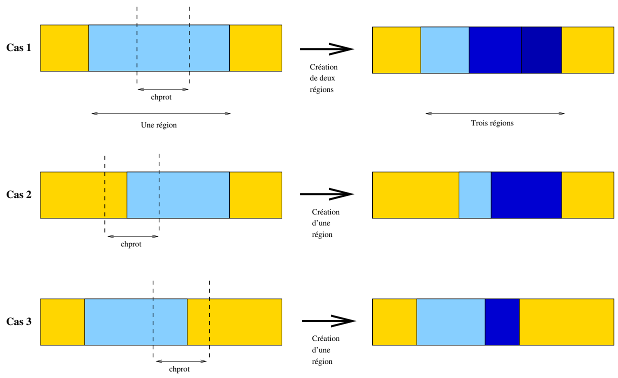

FIG. 17 – Illustration des trois différents cas rencontrés dans la fonction `sos_umem_vmm_chprot`

- **Cas 1 :** La région est de type privée et les nouveaux droits d'accès autorisent l'écriture. On ne peut pas simplement autoriser l'écriture dans les tables de pages, car il est possible que d'autres espaces d'adressage mappent la même ressource en privé, et ils ne doivent pas voir les modifications que l'on y applique. On ne modifie pas les tables de pages pour autoriser l'écriture. En effet, si la région était auparavant en lecture-seule, alors les pages sous-jacentes étaient aussi en lecture-seule, ce qui est parfait pour amorcer le mécanisme de COW. Sinon, la région était déjà en lecture/écriture et donc on n'a rien de spécial à modifier ;
- **Cas 2 :** Dans tous les autres cas, les nouveaux droits d'accès sont directement appliqués sur les tables de pages (appel à la fonction `sos_paging_set_prot_of_interval`).

Redimensionner une région virtuelle. La fonction `sos_umem_vmm_resize()` (variante du `mremap()` d'Unix) permet de redimensionner une région virtuelle. L'appelant doit donner l'adresse et la taille de la région actuelle, ainsi que l'adresse et la taille souhaitée pour cette région. Deux possibilités sont offertes à l'appelant : **1)** il n'accepte pas de déplacer la région à redimensionner, donc si le redimensionnement recouvre d'autres régions, il est refusé, **2)** il accepte de déplacer la région à redimensionner si le redimensionnement occasionne le recouvrement d'autres régions. Ce comportement est paramétrable à l'aide du drapeau `SOS_VR_REMAP_MAYMOVE`. Le paramètre `new_uaddr` est donc en entrée/sortie : en entrée il contient l'adresse de début souhaitée pour la région, en sortie il contient l'adresse réelle après redimensionnement.

Dans le cas où la région doit être déplacée, la ressource est simplement mappée à la nouvelle adresse (`sos_umem_vmm_map`) puis les pages physiques sont également remappées à cette nouvelle adresse. Enfin, la région initiale est supprimée (`sos_umem_vmm_unmap`).

Dans le cas où la région n'a pas besoin d'être déplacée, elle est simplement configurée avec les nouveaux paramètres.

2.4 Traitement des défauts de pages

Lorsqu'une exception de *défaut de page* est levée par le processeur, c'est toujours la fonction `pgflt_ex()` de `sos/main.c` qui est appelée. Celle-ci a été modifiée pour s'intégrer avec le nouveau système de gestion de mémoire virtuelle.

Accès depuis un thread en mode utilisateur. Désormais, lorsqu'un défaut de page a lieu dans un thread qui est en mode utilisateur, on tente de le résoudre en appelant la fonction `sos_umem_vmm_try_resolve_page_fault()`. Si celle-ci retourne `SOS_OK`, alors le défaut de page a été résolu. Sinon, le thread courant est détruit.

Cette fonction, implémentée dans `sos/umem_vmm.c` tente de résoudre un défaut de page en plusieurs étapes :

1. **Étape 1** : Récupération de la région virtuelle contenant l'adresse fautive en parcourant la liste des régions virtuelles de l'espace d'adressage courant. Si l'adresse fautive n'appartient à aucune région virtuelle, `-SOS_EFAULT` est renvoyé ;
2. **Étape 2** : Si le défaut de page est lié à une écriture sur une région en lecture seule, alors `-SOS_EFAULT` est renvoyé ;
3. **Étape 3** : Si le défaut de page est lié à une écriture sur une région en lecture/écriture, alors il faut réaliser un *Copy-on-write*. La fonction `sos_paging_try_resolve_COW()` étudiée précédemment est appelée pour cela ;
4. **Étape 4** : Le défaut de page est dans une région virtuelle valide et il ne s'agit pas d'un *Copy-on-write*. Il faut donc effectuer du *demand paging*, c'est-à-dire mapper à la demande le contenu de la région. C'est l'opération `page_in()` de la région qui permet de réaliser le *demand paging* : elle seule sait comment récupérer les données de la *ressource mappée*.

Cet algorithme permet donc de réaliser le demand paging, la copie-avant-écriture et de détecter les accès mémoire incorrects tant en lecture qu'en écriture. Mais il ne s'occupe pas de traiter le cas d'accès en exécution sur des régions marquées "non exécutable". C'est que les processeurs `x86` ne permettaient pas simplement, jusqu'à récemment, de déclencher une exception sur accès illicite en exécution. L'architecture `x86` est en train d'évoluer à ce sujet (nouveau bit NX d'AMD ou XD d'Intel dans les tables de pages).

Accès depuis un thread en mode noyau. Le mécanisme précédent est également déroulé si c'est un thread en mode noyau qui provoque le défaut de page. Il est donc possible de bénéficier de l'intégralité du mécanisme de COW et de demand paging depuis le noyau, ce qui est heureux. Prenons le cas du code noyau d'un appel système : rien n'empêche un thread utilisateur de fournir une donnée dans une région virtuelle parfaitement valide, mais à une adresse où la page virtuelle n'est pas mappée ou correspond à un mapping privé sur laquelle le noyau doit écrire (et ainsi amorcer le COW).

Ceci n'est cependant autorisé que si le thread noyau a demandé à s'exécuter dans son espace d'adressage nominal, c'est-à-dire celui du processus auquel il appartient, i.e. celui dans lequel il s'exécute quand il passe en mode utilisateur. Pour cela, nous utilisons le couple de fonctions `sos_thread_prepare_user_access()`/`sos_thread_end_user_access()` de `sos/thread.h`. Ces fonctions ont un rôle analogue à la fonction `sos_thread_change_current_mm_context()` que nous avions vue dans l'article 7. La première fonction du couple prend en paramètre l'espace d'adressage à "squatter", mais également une adresse. Cette adresse est stockée dans la structure `struct thread` et est utilisée par la routine de défaut de page quand elle n'arrive pas à résoudre le défaut de page (i.e. l'accès est en dehors des régions virtuelles ou possède les mauvais droits). Dans ce cas, mieux vaut ne pas faire comme dans le cas des threads utilisateur (arrêter le thread) : mieux vaut être capable de continuer de fonctionner, tout en signalant le problème quand on reviendra en mode utilisateur. C'est précisément l'adresse précédente qui indique là où la routine de traitement du défaut de page doit retourner pour que le noyau puisse prendre les mesures "exceptionnelles" nécessaires.

Prenons par exemple un extrait du code d'une des fonctions de `sos/uaccess.c`, `sos_memcpy_from_user()`, qui permet au noyau de récupérer des données depuis l'espace utilisateur :

```
  __label__ catch_pgflt;

  sos_thread_prepare_user_space_access(NULL,
                                       (sos_vaddr_t) && catch_pgflt);

  for (cptr_dst = (char*)dest,
         cptr_src = (const char*)src,
         transfer_sz = size ;
       transfer_sz > 0 ;
       cptr_dst++, cptr_src++, transfer_sz--)
    *cptr_dst = *cptr_src;

  sos_thread_end_user_space_access();
  return size;

catch_pgflt:
  {
    sos_uaddr_t faulted_uaddr = cur_thr->fixup_uaccess.faulted_uaddr;
    sos_thread_end_user_space_access();
    return faulted_uaddr - src;
  }
```

Dans cet exemple, la boucle `for(;;)` peut déclencher des défauts de page sans problème. Le demand paging sera pris en charge, le COW aussi[2]. Mais en cas d'accès interdit, la routine d'exception demandera à redémarrer le code au label `catch_pgflt` qui s'occupera de traiter le cas exceptionnel. La routine de traitement du défaut de page aura auparavant stocké l'adresse où la faute insoluble s'est produite dans le champ `fixup_uaccess.faulted_uaddr` du thread. On remarquera au passage la forme, propre à `gcc`, pour récupérer l'adresse d'un label local ("`&& nom_label`").

Pour information, Linux repose sur une méthode relativement proche. Sauf qu'il utilise un dictionnaire pour savoir à quelle adresse retourner quand un défaut de page insoluble s'est produit à telle adresse.

2.5 Pilote "`zero`"

Le pilote `zero` permet de projeter en mémoire un faux fichier dont la taille serait infinie et le contenu entièrement à zéro. Il permet donc de créer des régions virtuelles initialisées à zéro. Il fonctionne de manière similaire à `/dev/zero` sous Unix (et aussi à `/dev/null` en ce qui concerne `mmap`).

Ce pilote est implémenté dans le fichier `drivers/zero.c`. La ressource mappée est représentée par une structure `zero_mapped_resource`. Celle-ci contient la structure `sos_umem_vmm_mapped_resource` nécessaire au système de gestion de mémoire virtuelle, ainsi qu'un compteur de référence. Ce dernier permettra de libérer la ressource lorsqu'elle deviendra inutilisée.

[2] En réalité, pour que le COW sur des pages utilisateur puisse avoir lieu quand le processeur est en mode superviseur, il faut que le bit WP du registre `cr0` soit positionné [8, section 4.11.3]. Dans SOS, ce bit avait été positionné dans `hwcore/paging.c` (article 4).

Le cœur du pilote est constitué des opérations de la structure `sos_umem_vmm_vr_ops`. Notre pilote en implémente trois :

– **ref**, implémenté dans `zero_ref()` est appelé à chaque fois que la ressource est mappée dans un espace d'adressage. Elle permet de maintenir le compteur de références sur la ressource ;
– **unref**, implémenté dans `zero_unref()` est appelé à chaque fois que la ressource est démappée d'un espace d'adressage. Elle permet également de maintenir le compteur de références sur la ressource. Si celui-ci descend jusqu'à 0, alors la ressource est supprimée ;
– **page_in**, implémenté dans `zero_page_in()` est appelé pour effectuer le *demand paging* lorsqu'un défaut de page sur une région mappant une ressource *zéro* a été détecté. Si le mapping est de type "partagé", la première chose que fait **page_in** est de rechercher une page qui aurait déjà été mappée pour le même emplacement dans la ressource par d'autres régions virtuelles : un dictionnaire propre à la ressource est maintenu pour cela. Si une telle page est trouvée, c'est elle qu'on mappe et on a fini. Sinon, si le défaut de page est lié à un accès en écriture, alors une nouvelle page est allouée et mappée dans l'espace d'adressage ; on n'oublie pas d'enregistrer cette page allouée dans le dictionnaire précédent si le mapping est de type shared. Si le défaut de page est lié à un accès en lecture, alors une page spéciale `sos_zero_page` commune à tout le système et remplie de 0 (allouée par `umem_vmm.c`) est mappée en lecture seule. Il s'agit ici d'une petite optimisation. Puisque les pages mappées par notre pilote sont initialisées à zéro, tant que des accès en lecture sont effectués, on peut partager cette page spéciale. Celle-ci est mappée en lecture seule, de manière à ce que le premier accès en écriture déclenche un défaut de page qui permettra d'allouer réellement une page (mécanisme de COW).

La fonction `sos_zero_map()` permet de créer une nouvelle région basée sur le pilote `zero`, en spécifiant un espace d'adressage, une adresse, une taille, des droits d'accès et des drapeaux. Après avoir alloué et initialisé la structure `zero_mapped_resource`, la région virtuelle est effectivement créée par un appel à `sos_umem_vmm_map()`. La fonction `zero_mmap()` sera appelée par le système `umem_vmm` et sert à renseigner ce dernier des fonctions à utiliser pour les méthodes *ref*, *unref* et *page_in*.

2.6 Chargement des applications utilisateur

SOS ne dispose toujours pas de pilote de disque dur, et donc pas de système de fichiers avec stockage. Comme dans l'article 7, les applications utilisateur sont donc stockées au sein de l'image du noyau elle-même. Toutefois, un chargeur ELF simple a été implémenté dans le fichier `sos/binfmt_elf32.c`. Il permet de charger une application dans un espace d'adressage en créant toutes les régions virtuelles qui sont nécessaires. Il implémente également la `mapped_resource` permettant la projection en mémoire de ces applications. En réalité, ces applications étant chargées avec le noyau, il permet de remapper dans une région virtuelle une partie de l'espace noyau. Une telle ressource est représentée par une structure `elf32_mapped_program`.

Le cœur du chargeur se situe dans la fonction `sos_binfmt_elf32_map()` qui permet de charger une application dans un espace d'adressage donné. Un nom est donné à chaque application, qu'on peut associer à un nom de fichier dans un système d'exploitation complet. Elle commence par rechercher un programme possédant un tel nom, grâce à la fonction `lookup_userprog()`. Cette dernière repose sur les structures de données construites par les scripts de *linkage* décrits de manière extensive dans l'article 7. Une fois l'adresse du programme déterminée, la structure `elf32_mapped_program` est initialisée, et le chargement du fichier ELF proprement dit commence. Tout d'abord, quelques vérifications sont effectuées, puis pour chaque entrée du *program header* ELF32 marquée du drapeau *LOAD*, une région virtuelle est créée à partir de la *mapped resource*. L'offset de la zone de la ressource mappée par la région virtuelle est calculé grâce aux informations du *program header*.

Le reste du pilote est constitué de l'implémentation des méthodes *ref*, *unref* et *page_in*. Les deux premières sont relativement triviales. La dernière, `elf32prog_page_in()` est un peu plus délicate. Elle commence par déterminer à quel déplacement dans la ressource se situe les informations à charger dans la page, puis elle alloue et mappe une nouvelle page en mémoire avant d'y copier les informations de la ressource. Ici, la ressource étant présente dans l'espace noyau, il s'agit simplement de recopier des données depuis un autre endroit de la mémoire. Dans certains cas favorables (condition d'alignement sur des frontières de pages) on peut "optimiser" en partageant la page directement avec le noyau, sans copier son contenu ni allouer d'autres pages physiques : on remappe la page noyau directement en espace utilisateur (voir le code de SOS). Cette optimisation est facultative bien entendu.

Lorsque SOS sera un système plus complet, la méthode *page_in* se chargera d'accéder au système de fichiers, puis de recopier en mémoire la partie du fichier mappé.

2.7 Duplication de threads

L'implémentation de `fork()` nécessite non seulement de recopier l'espace d'adressage, mais également le thread du processus père qui a fait le `fork()`. Il faut en effet que le nouveau thread ait exactement les mêmes valeurs pour tous les registres afin d'accéder aux données aux mêmes adresses virtuelles que le thread parent et afin de reprendre son exécution là où en était le parent. Un seul registre sera différent entre le père et le fils pour qu'ils puissent se différencier : c'est le registre qui stocke la valeur de retour du `fork` (`eax`, par convention). Les éventuels autres threads du processus parent ne sont pas recopiés dans le fils.

Le problème de la recopie de l'espace d'adressage a déjà été évoqué (2.3.2), il est effectué grâce à la fonction `sos_umem_vmm_duplicate_as()`. En revanche, celui de la recopie des threads n'a pas encore été traité.

La fonction `create_user_thread()` de `sos/thread.c` a été modifiée pour prendre un paramètre supplémentaire : `model_thread`. Lorsque celui-ci est nul, la fonction créé simplement un nouveau thread utilisateur (voir l'article 7). Mais lorsque celui-ci n'est pas nul, alors il est utilisé comme modèle pour la création d'un nouveau thread. Pour initialiser le contexte de ce nouveau thread, une nouvelle fonction `sos_cpu_ustate_duplicate()` est utilisée. Contrairement à `sos_cpu_ustate_init()`, elle initialise le contexte par recopie de celui d'un autre thread.

Ces fonctions sont implémentées dans `hwcore/cpu_context.c`. Leurs grandes similitudes ont permis de les factoriser dans une fonction appelée `cpu_ustate_init()`. Dans le cas où le contexte est un nouveau contexte, il est initialisé à zéro, puis l'adresse de début du programme `EIP` et du haut de la pile `ESP` sont initialisées grâce aux valeurs passées en paramètre. Dans le cas où le contexte est une recopie d'un autre, alors un simple `memcpy()` est utilisé. Seule la valeur du registre `EAX` est modifiée (positionnée à la valeur du paramètre `user_retval` passé à `sos_cpu_ustate_duplicate`) : cela permet à `fork()` de "renvoyer" des valeurs différentes pour le processus fils et le processus père.

2.8 Mécanismes de haut-niveau pour la création de processus

Les mécanismes de création de processus mis en place à l'article 7 ont été améliorés de manière à prendre en compte les nouvelles possibilités de gestion de la mémoire virtuelle.

En particulier, la fonction `sos_process_create()` prend désormais un nouvel argument, le booléen `do_copy_current_process`, qui vaut `TRUE` si on souhaite que le nouveau processus ait une copie de l'espace d'adressage du processus courant (en utilisant `sos_umem_vmm_duplicate_as()`).

La nouvelle fonction `sos_process_set_address_space()` permet de modifier l'espace d'adressage associé à un processus. Si un espace d'adressage pré-existe pour ce processus, alors il est supprimé. Cette primitive est utile pour l'implémentation de l'appel système `exec`.

2.9 Implémentation des appels système

Les appels système `fork` et `exec` sont implémentés dans le fichier `sos/syscall.c` à partir de toutes les primitives étudiées dans cet article. Ils sont présents au sein de la fonction `sos_do_syscall()`.

2.9.1 `fork`

Implémentation. L'implémentation de `fork` consiste simplement à créer un nouveau processus en prenant pour modèle le processus courant (appel à `sos_process_create`), puis à dupliquer le thread ayant réalisé l'appel système (appel à `sos_duplicate_user_thread`). Un graphe d'appel simplifié de cet appel système est proposé en figure 18. Il résume l'ensemble des opérations réalisées et des sous-systèmes impliquées dans un `fork`.

Scénario d'exécution. Pour illustrer le fonctionnement de `fork`, nous vous proposons d'imaginer un petit scénario simple. Soit un processus A dans lequel sont présentes quatre régions virtuelles :

- une région virtuelle privée en lecture seule, contenant le code ;
- une région virtuelle privée en lecture/écriture, contenant les données ;
- une région virtuelle partagée anonyme en lecture/écriture, contenant des données qui doivent être partagées avec les fils ;

FIG. 18 – Graphe d'appel simplifié de l'appel système `fork`

- une région virtuelle privée en lecture/écriture, contenant la pile.

Ce processus A effectue un `fork`, donnant naissance au processus B. Durant le `fork`, l'ensemble des régions virtuelles de A est dupliqué dans B, et les régions virtuelles privées en lecture/écriture (les données et la pile) sont passées en lecture seule pour permettre le fonctionnement du *Copy-on-write*.

Suite au `fork`, le processus A et le processus B reprennent l'exécution de leur code (similaire dans les deux processus). Ce code étant dans une région en lecture seule, il a été mappé en lecture seule dans les deux processus. Son exécution par les deux processus n'engendre donc aucun défaut de page.

Au cours de l'exécution du code, les deux processus A et B ont besoin d'empiler des informations sur leur pile (plus exactement, c'est un thread de A et un thread de B qui le font). Les piles sont des régions virtuelles privées en lecture/écriture. Lors du `fork`, ces régions ont été passées en lecture seule pour permettre le *Copy-on-write*. Si par exemple A est la première à accéder en écriture à la pile, un défaut de page sera généré. S'agissant d'un défaut de page en écriture sur une page en lecture seule appartenant à une région en lecture/écriture, le gestionnaire de défaut de page déclenchera le *Copy-on-write*. La page physique correspondante ayant un compteur de référence à 2 (partagée par A et B), il effectuera une copie dans une nouvelle page, qu'il mappera en lecture/écriture dans le processus A. Ce dernier pourra donc poursuivre son exécution. Un peu plus tard, B s'exécute, et a donc besoin d'écrire sur sa pile. Or, celle-ci étant toujours mappée en lecture seule, un défaut de page est donc généré. De la même manière que pour A, le mécanisme de *Copy-on-write* est déclenché. Cette fois-ci, en revanche, la page physique correspondante ayant un compteur de référence à 1 (utilisée seulement par B), il suffira de la passer en lecture/écriture pour B qui pourra poursuivre son exécution.

Pour la région contenant les données, le mécanisme sera identique : tant que A et B accéderont à une page en lecture uniquement, elle restera partagée. Au premier accès en écriture, il y aura *Copy-on-write*.

Lorsque A voudra transmettre des informations à B au travers de la région anonyme partagée, il lui suffira d'écrire dedans. Celle-ci étant en lecture/écriture et partagée, le `fork` a laissé le mapping des pages en lecture/écriture, et a simplement recopié ces mappings dans le nouveau processus B. Ainsi, toute écriture de A sera automatiquement vue par B, et vice-versa.

2.9.2 `exec`

Implémentation. L'implémentation de `exec` commence par récupérer les deux arguments de l'appel système : la chaîne de caractères contenant le nom du programme à exécuter et sa taille. Après avoir recopié cette chaîne de caractères dans un espace mémoire du noyau, la fonction créé un nouvel espace d'adressage vide pour le processus courant (appel à `sos_umem_vmm_create_empty_as`). Le programme à exécuter est ensuite mappé dans ce nouvel espace d'adressage grâce au chargeur ELF puis une pile utilisateur de 8 Mo est mappée en mémoire (mapping anonyme privé). Enfin, un nouveau thread utilisateur est créé pour exécuter le programme et l'espace d'adressage nouvellement créé est attaché au processus courant (appel à `sos_process_set_address_space`). Cette dernière opération prendra en charge la destruction de l'ancien espace d'adressage. La figure 19 présente un graphe d'appel simplifié de l'appel système `exec`.

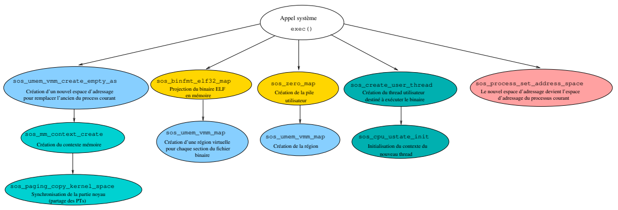

FIG. 19 – Graphe d'appel simplifié de l'appel système `exec`

Scénario d'exécution. Un processus A contenant trois régions (une région de code, une région de données, une région de pile) effectue un appel `exec` pour exécuter une application donnée. Grâce au chargeur de fichiers ELF, Le système commence par rechercher cette application, puis à l'aide de son entête ELF, détermine sa structure. Après avoir vidé le processus A de ses anciennes régions virtuelles, il en créé de nouvelles correspondant à la nouvelle application. Rien n'est chargé en mémoire, seules les régions virtuelles sont créées.

Lorsque le processus commence à exécuter le code nouvellement chargé, un défaut de page est immédiatement levé. En effet, aucune page de la région de code n'est mappé. Le gestionnaire de défaut de page s'en aperçoit, et appelle l'opération `page_in` (du pilote `binfmt`, puisqu'il s'agit d'une région associée à une ressource gérée par ce pilote). Celle-ci alloue une page physique, la mappe à la bonne adresse virtuelle puis y copie la portion du code de l'application correspondant à la page fautive. Le processus peut ainsi commencer réellement son exécution.

Rapidement, il va devoir utiliser la pile, une région anonyme gérée par le pilote `zero`. Aucune page n'étant mappée, un défaut de page est levé, et l'opération `page_in` du pilote `zero` est appelé. Celui-ci alloue une page physique, la mappe en mémoire et l'initialise à zéro. Le processus peut alors utiliser la pile, et poursuivre son exécution.

Les mêmes mécanismes sont utilisés pour traiter chaque défaut de page. Ainsi, l'application n'est pas chargée en mémoire au moment de l'appel à `exec` : des portions de celles-ci sont chargées au fur et à mesure des défauts de page, lorsque c'est nécessaire.

2.9.3 Autres appels système

On implémente les appels système Unix classiques relatifs à la gestion de l'espace virtuel utilisateur : `brk`, `mmap`, `munmap`, `mprotect` et `mremap`. Se reporter aux pages du manuel Unix pour les comprendre. Nous en avons profité pour en rajouter toute une série, non standard mais bien commentée, consacrée à la gestion/synchronisation des threads utilisateur (nous aurions pu faire ceci dès l'article 7).

La différence avec Unix est que pour l'instant la ressource qu'on passe à `mmap` n'est pas un "file descriptor" (nous verrons cette notion dans le prochain article). Il s'agit d'une chaîne de caractères représentant la ressource à mapper. Les chaînes de caractères reconnues pour le moment sont :

- `"/dev/zero"` : mapping anonyme (source : `drivers/zero.c`),
- `"/dev/mem"` : mapping de la mémoire physique (source : `drivers/mem.c`),
- `"/dev/kmem"` : mapping de l'espace noyau (source : `drivers/kmem.c`).

Les sources des deux derniers "pilotes" peuvent d'ailleurs servir de modèles simples pour qui veut implanter son propre pilote de ressource mappable.

3 La démo du mois

Cet article propose une initialisation du système similaire à celle que l'on peut trouver dans un système Unix : un processus initial `init` est créé *ex-nihilo* par le noyau à partir du programme `userland/init`. C'est l'unique processus créé de cette manière par le noyau, les autres le seront par `fork` et `exec` à partir de ce processus initial. Dans SOS, le processus `init` est créé dans la fonction `start_init()` de `sos/main.c`.

Il effectue toute une série de tests :

- création et modifications de mappings anonymes partagés,
- création et modifications de mappings anonymes privés,
- appels à `munmap()`,
- fonction `malloc()` primitive qui appelle `brk()`,
- `fork()` suivi d'accès aux mappings partagées/privés,
- `exec()` des autres programmes de `userland/`,
- appels système déclenchant des défauts de pages licites et illicites côté noyau pendant les accès à la zone utilisateur (voir la section 2.4),
- mapping de la mémoire vidéo `x86` en mode partagé pour l'affichage direct à l'écran depuis un processus utilisateur.

Malheureusement la copie d'écran n'est pas très probante. En dehors de servir de test, le fichier `userland/init.c` a le grand mérite de pouvoir servir de point de départ pour écrire ses propres processus SOS !

Conclusion

Au cours de cet article, nous avons pu mettre en place un système de gestion de mémoire virtuelle minimal pour l'espace utilisateur. Celui-ci permet de projeter en mémoire des ressources au sein de régions virtuelles (`mmap`) que l'on peut redimensionner (`mremap`) ou dont on peut changer les droits d'accès (`mprotect`). Cette projection en mémoire est

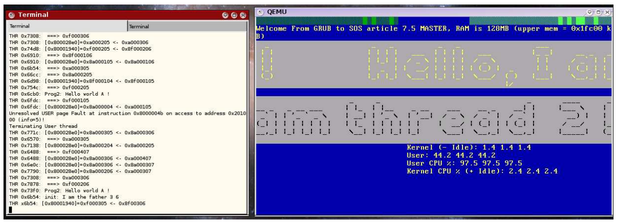

FIG. 20 – Aperçu de la petite démo (à droite : terminal pour observer les sorties sur le port `0xe9`)

basée sur la gestion des défauts de page, utilisant en particulier les mécanismes de *demand-paging* et de *Copy-on-write*. Sur cette base, il a été possible de construire quelques appels système importants, dont `fork` et `exec`.

Du point de vue de la gestion de la mémoire et de la gestion des processus, SOS dispose maintenant des fonctionnalités indispensables. Des améliorations sont bien entendu possibles, comme par exemple l'implémentation de mécanismes de communication inter-processus. Toutefois, les prochains articles se consacreront à un autre sujet : les systèmes de fichiers. En particulier, l'article 8 permettra d'étudier la mise en place d'un système de fichier virtual (*Virtual Filesystem*), l'article 9 permettra d'implémenter un pilote de disque dur tandis que l'article 10 permettra d'implémenter un pilote de système de fichiers simple, afin de charger les programmes utilisateur depuis de *vrais* fichiers !

Pour terminer, répétons que le code de ce mois-ci n'est pas disponible sur le CDROM accompagnant l'article : il est mis sur le site de SOS (`http://sos.enix.org`). Profitons-en pour rappeler que vous pourrez trouver sur ce site le code et le texte des articles précédents ainsi que les informations pour vous inscrire à la mailing-list ou pour lire les archives de ladite liste.

The end.

Thomas Petazzoni et David Decotigny
`thomas.petazzoni@enix.org` et `d2@enix.org`
Merci à Nessie pour sa relecture, ses remarques constructives et ses propositions.
Site de SOS : `http://sos.enix.org`
Projet KOS : `http://kos.enix.org`
Á Camille

Références

[1] Mel Gorman. Understanding the Linux virtual memory manager.
`http://www.skynet.ie/~mel/projects/vm/guide/html/understand/`, 2004.

[2] Charles D. Cranor. *Design and implementation of UVM.*
`http://www.ccrc.wustl.edu/pub/chuck/psgz/diss.ps.gz`.

[3] Charles D. Cranor et Gurudatta M. Parulkar. The uvm virtual memory system.
`http://www.usenix.org/events/usenix99/full_papers/cranor/cranor.pdf`, 1999.

[4] Joseph P. Moran. Sunos virtual memory implementation.
`http://citeseer.ist.psu.edu/moran88sunos.html`, 1988.

[5] William A. Shannon Robert A. Gingell, Joseph P. Moran. Virtual memory architecture in sunos.
`http://www.solarisinternals.com/si/reading/vm-arch.pdf`, 1987.

[6] Marc Shapiro Vadim Abrossimov, Marc Rozier. Generic virtual memory management for operating system kernels (chorus).
`http://citeseer.ist.psu.edu/abrossimov89generic.html`, 1989.

[7] TIS Committee. TIS ELF.
`www.x86.org/ftp/manuals/tools/elf.pdf`, 1995.

[8] Intel Corp. Intel architecture developer's manual, vol 3, 1997.